CHAPTER 8

Using the Session

You may have noticed that the `Session` object is the central point of access to Hibernate functionality. We will now look at what it embodies and what that implies about how you should use it.

Sessions

From the examples in the earlier chapters, you will have noticed that a small number of classes dominate our interactions with Hibernate. Of these, `Session` is the linchpin.

The `Session` object is used to create new database entities, read in objects from the database, update objects in the database, and delete objects from the database. It allows you to manage the transaction boundaries of database access, and (in a pinch) it allows you to obtain a traditional JDBC connection object so that you can do things to the database that the Hibernate developers have not already considered in their existing design (precious little).

If you are familiar with the JDBC approach, it helps to think of a `Session` object as being somewhat like a JDBC connection, and the `SessionFactory`, which provides `Session` objects, as being somewhat like a `ConnectionPool`, which provides `Connection` objects. These similarities in roles are illustrated in Figure 8-1.

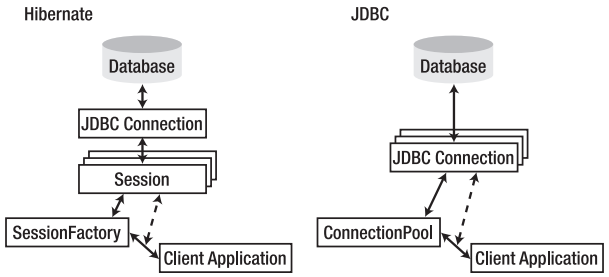

Figure 8-1. *Similarities between Hibernate and JDBC objects*

`SessionFactory` objects are expensive objects—needlessly duplicating them will cause problems quickly, and creating them is a relatively time-consuming process. Ideally, you should have a single `SessionFactory` for each database your application will access. `SessionFactory` objects are threadsafe, so it is not necessary to obtain one for each thread. However, you will create numerous `Session` objects—at least one for each thread using Hibernate. Sessions in Hibernate are *not* threadsafe, so sharing `Session` objects between threads could cause data loss or deadlock. In fact, you will often want to create multiple `Session` instances even during the lifetime of a specific thread (see the "Threads" section for concurrency issues).

Caution The analogy between a Hibernate session and a JDBC connection only goes so far. One important difference is that if a Hibernate `Session` object throws an exception of any sort, you must discard it and obtain a new one. This prevents data in the session's cache from becoming inconsistent with the database.

We've already covered the core methods in Chapter 4, so we won't discuss all the methods available to you through the `Session` interface. For an exhaustive look at what's available, you should read the API documentation on the Hibernate web site or in the Hibernate 3 download. Table 8-1 gives an overview of the various categories of methods available to you.

Table 8-1. *Hibernate Method Summary*

Method	Description
Create, Read, Update, and Delete	
`save()`	Saves an object to the database. This should not be called for an object that has already been saved to the database.
`saveOrUpdate()`	Saves an object to the database, or updates the database if the object already exists. This method is slightly less efficient than the `save()` method since it may need to perform a `SELECT` statement to check whether the object already exists, but it will not fail if the object has already been saved.
`merge()`	Merges the fields of a nonpersistent object into the appropriate persistent object (determined by ID). If no such object exists in the database, then one is created and saved.
`persist()`	Reassociates an object with the session so that changes made to the object will be persisted.
`get()`	Retrieves a specific object from the database by the object's identifier.
`getEntityName()`	Retrieves the entity name (this will usually be the same as the class name of the POJO).
`getIdentifier()`	Determines the identifier—the object(s) representing the primary key—for a specific object associated with the session.
`load()`	Loads an object from the database by the object's identifier (you should use the `get()` methods if you are not certain that the object is in the database).
`refresh()`	Refreshes the state of an associated object from the database.
`update()`	Updates the database with changes to an object.
`delete()`	Deletes an object from the database.
`createFilter()`	Creates a filter (query) to narrow operations on the database.

Method	Description
`enableFilter()`	Enables a named filter in queries produced by `createFilter()`.
`disableFilter()`	Disables a named filter.
`getEnabledFilter()`	Retrieves a currently enabled filter object.
`createQuery()`	Creates a Hibernate query to be applied to the database.
`getNamedQuery()`	Retrieves a query from the mapping file.
`cancelQuery()`	Cancels execution of any query currently in progress from another thread.
`createCriteria()`	Creates a criteria object for narrowing search results.
Transactions and Locking	
`beginTransaction()`	Begins a transaction.
`getTransaction()`	Retrieves the current transaction object. This does not return `null` when no transaction is in progress. Instead, the active property of the returned object is `false`.
`lock()`	Gets a database lock for an object (or can be used like `persist()` if `LockMode.NONE` is given).
Managing Resources	
`contains()`	Determines whether a specific object is associated with the database.
`clear()`	Clears the session of all loaded instances and cancels any saves, updates, or deletions that have not been completed. Retains any iterators that are in use.
`evict()`	Disassociates an object from the session so that subsequent changes to it will not be persisted.
`flush()`	Flushes all pending changes into the database—all saves, updates, and deletions will be carried out; essentially, this synchronizes the session with the database.
`isOpen()`	Determines whether the session has been closed.
`isDirty()`	Determines whether the session is synchronized with the database.
`getCacheMode()`	Determines the caching mode currently employed.
`setCacheMode()`	Changes the caching mode currently employed.
`getCurrentLockMode()`	Determines the locking mode currently employed.
`setFlushMode()`	Determines the approach to flushing currently used. The options are to flush after every operation, flush when needed, never flush, or flush only on commit.
`setReadOnly()`	Marks a persistent object as read-only (or as writable). There are minor performance benefits from marking an object as read-only, but changes to its state will be ignored until it is marked as writable.
`close()`	Closes the session, and hence, the underlying database connection; releases other resources (such as the cache). You must not perform operations on the `Session` object after calling `close()`.
`getSessionFactory()`	Retrieves a reference to the `SessionFactory` object that created the current `Session` instance.
The JDBC Connection	
`connection()`	Retrieves a reference to the underlying database connection.
`disconnect()`	Disconnects the underlying database connection.
`reconnect()`	Reconnects the underlying database connection.
`isConnected()`	Determines whether the underlying database connection is connected.

Transactions and Locking

Transactions and locking are intimately related—the locking techniques chosen to enforce a transaction can determine both the performance and likelihood of success of the transaction. The type of transaction selected dictates, to some extent, the type of locking that it must use.

You are not obliged to use transactions if they do not suit your needs, but there is rarely a good reason to avoid them. If you decide to avoid them, you will need to invoke the `flush()` method on the session at appropriate points to ensure that your changes are persisted to the database.

Transactions

A transaction is a unit of work guaranteed to behave as if you have exclusive use of the database. If you wrap your work in a transaction, the behavior of other system users will not affect your data. A transaction can be started, committed to write data to the database, or rolled back to remove all changes from the beginning onward (usually as the result of an error). To achieve this, you obtain a `Transaction` object from the database (beginning the transaction) and manipulate the session as shown in the following code:

```
Session session = factory.openSession();
try {
  session.beginTransaction();

  // Normal session usage here...

  session.getTransaction().commit();
} catch (HibernateException e) {
  Transaction tx = session.getTransaction();
  if (tx.isActive()) tx.rollback();
} finally {
  session.close();
}
```

In the real world, it's not actually desirable for all transactions to be fully ACID (see the sidebar entitled "The ACID Tests") because of the performance problems that this can cause.

Different database suppliers support and permit you to break the ACID rules to a lesser or greater extent, but the degree of control over the isolation rule is actually mandated by the SQL-92 standard. There are important reasons that you might want to break this rule, so both JDBC and Hibernate also make explicit allowances for it.

THE ACID TESTS

- *Atomicity*: A transaction should be all or nothing. If it fails to complete, the database will be left as if none of the operations had ever been performed—this is known as a *rollback*.
- *Consistency*: A transaction should be incapable of breaking any rules defined for the database. For example, foreign keys must be obeyed. If for some reason this is impossible, the transaction will be rolled back.
- *Isolation*: The effects of the transaction will be completely invisible to all other transactions until it has completed successfully. This guarantees that the transaction will always see the data in a sensible state. For example, an update to a user's address should only contain a correct address (i.e., it will never have the house name for one location but the ZIP code for another); without this rule, a transaction could easily see when another transaction had updated the first part but had not yet completed.
- *Durability*: The data should be retained intact. If the system fails for any reason, it should always be possible to retrieve the database up to the moment of the failure.

The isolation levels permitted by JDBC and Hibernate are listed in Table 8-2.

Table 8-2. *JDBC Isolation Levels*

Level	Name	Transactional Behavior
0	None	Anything is permitted; the database or driver does not support transactions.
1	Read Uncommitted	Dirty, nonrepeatable, and phantom reads are permitted.
2	Read Committed	Nonrepeatable reads and phantom reads are permitted.
4	Repeatable Read	Phantom reads are permitted.
8	Serializable	The rule must be obeyed absolutely.

A *dirty read* may see the in-progress changes of an uncommitted transaction. As with the isolation example discussed in the preceding sidebar, it could see the wrong ZIP code for an address.

A *nonrepeatable read* sees different data for the same query. For example, it might determine a specific user's ZIP code at the beginning of the transaction and again at the end, and get a different answer both times without making any updates.

A *phantom read* sees different numbers of rows for the same query. For example, it might see 100 users in the database at the beginning of the query and 105 at the end without making any updates.

The HSQLDB database that we are using in this book only supports the first level of isolation here: Read Uncommitted. While this means that deadlocks cannot occur (see the "Deadlocks" section later in the chapter), the three undesirable behaviors of dirty, nonrepeatable, and phantom reads are permitted.

Hibernate treats the isolation as a global setting—you apply the configuration option `hibernate.connection.isolation` in the usual manner, setting it to one of the values permitted in Table 8-2. This is not always ideal. You will sometimes want to treat one particular transaction at a high level of isolation (usually Serializable), while permitting lower degrees of isolation for others. To do so, you will need to obtain the JDBC connection directly, alter the isolation level, begin the transaction, roll back or clean up the transaction as appropriate, and reset the isolation level back to its original value before releasing the connection for general usage. Hibernate does not provide a more direct way to alter the isolation level of the connection in a localized way. The implementation of the `createUser()` method, shown in Listing 8-1, demonstrates the additional complexity that the connection-specific transaction isolation involves.

Listing 8-1. *Using a Specific Isolation Level*

```
public static void createUser(String username)
   throws HibernateException
{
   Session session = factory.openSession();
   int isolation = -1;
   try {
      isolation = session.connection().getTransactionIsolation();
      session.connection().setTransactionIsolation(
         Connection.TRANSACTION_SERIALIZABLE);
      session.beginTransaction();

      // Normal usage of the Session here...
      Publisher p = new Publisher(username);
      Subscriber s = new Subscriber(username);
      session.saveOrUpdate(p);
      session.saveOrUpdate(s);

      // Commit the transaction
      session.getTransaction().commit();
   } catch (SQLException e1) {
      rollback(session);
      throw new HibernateException(e1);
   } catch (HibernateException e1) {
      rollback(session);
      throw e1;
   } finally {
      // reset isolation
      reset(session,isolation);

      // Close the session
      close(session);
   }
}
```

Fortunately, the normal case for a transaction using the global isolation level is much simpler. We provide a more standard implementation of the `createUser()` method for comparison in Listing 8-2.

Listing 8-2. *Using the Global (Default) Isolation Level*

```
public static void createUser(String username) throws HibernateException {
   Session session = factory.openSession();
   try {
      session.beginTransaction();

      // Normal usage of the Session here...
      Publisher p = new Publisher(username);
      Subscriber s = new Subscriber(username);
      session.saveOrUpdate(p);
      session.saveOrUpdate(s);

      // Commit the transaction
      session.getTransaction().commit();
   } catch (HibernateException e1) {
      rollback(session);
      throw e1;
   } finally {
      // Close the session
      close(session);
   }
}
```

Locking

A database can conform to these various levels of isolation in a number of ways, and you will need a working knowledge of locking to elicit the desired behavior and performance from your application in all circumstances.

To prevent simultaneous access to data, the database itself will acquire a lock on that data. This can be acquired for the momentary operation on the data only, or it can be retained until the end of the transaction. The former is called *optimistic locking* and the latter is called *pessimistic locking*.

The Read Uncommitted isolation level always acquires optimistic locks, whereas the Serializable isolation level will only acquire pessimistic locks. Some databases offer a feature that allows you to append the `FOR UPDATE` query to a select operation, which requires the database to acquire a pessimistic lock even in the lower isolation levels.

Hibernate provides some support for this feature when it is available, and takes it somewhat further by adding facilities that describe additional degrees of isolation obtainable from Hibernate's own cache.

The `LockMode` object controls this fine-grained isolation (see Table 8-3). It is only applicable to the `get()` methods, so it is limited—however, when possible, it is preferable to the direct control of isolation mentioned previously.

Table 8-3. *Lock Modes*

Mode	Description
`NONE`	Reads from the database only if the object is not available from the caches.
`READ`	Reads from the database regardless of the contents of the caches.
`UPGRADE`	Obtains a dialect-specific upgrade lock for the data to be accessed (if this is available from your database).
`UPGRADE_NOWAIT`	Behaves like `UPGRADE`, but when support is available from the database and dialect, the method will fail with a locking exception immediately. Without this option, or on databases for which it is not supported, the query must wait for a lock to be granted (or for a timeout to occur).

An additional lock mode, `WRITE`, is acquired by Hibernate automatically when it has written to a row within the current transaction. This mode cannot be set explicitly, but calls to `getLockMode` may return it.

Having discussed locking in general, we need to touch on some of the problems that locks can cause.

Deadlocks

Even if you have not encountered a deadlock (sometimes given the rather louche name of "deadly embrace") in databases, you have probably encountered the problem in multithreaded Java code. The problem arises from similar origins.

Two threads of execution can get into a situation in which each is waiting for the other to release a resource that it needs. The most common way to create this situation in a database is shown in Figure 8-2.

Each thread obtains a lock on its table when the update begins. Each thread proceeds until the table held by the other user is required. Neither thread can release the lock on its own table until the transaction completes—so something has to give.

A deadlock can also occur when a single thread of execution is carrying out an equivalent sequence of operations using two `Session` objects connected to the same database. In practice, the multiple-thread scenario is more common.

Fortunately, a database management system (DBMS) can detect this situation automatically, at which point the transaction of one or more of the offending processes will be aborted by the database. The resulting deadlock error will be received and handled by Hibernate as a normal `HibernateException`. Now you must roll back your transaction, close the session, and then (optionally) try again.

Listing 8-3 demonstrates how four updates from a pair of sessions can cause a deadlock. If you look at the output from the threads, you will see that one of them completes while the other fails with a deadlock error.

■ **Caution** The HSQL database does not support sufficient levels of isolation to run this example—you will never get a deadlock, and you will get inconsistent data. If you want to see the deadlock in action, you will need to use a more sophisticated database product (e.g., PostgreSQL).

You should also be aware that if you want to run the test using the MySQL database, you must use `MySQLInnoDBDialect` to ensure that the appropriate level of transactions is supported.

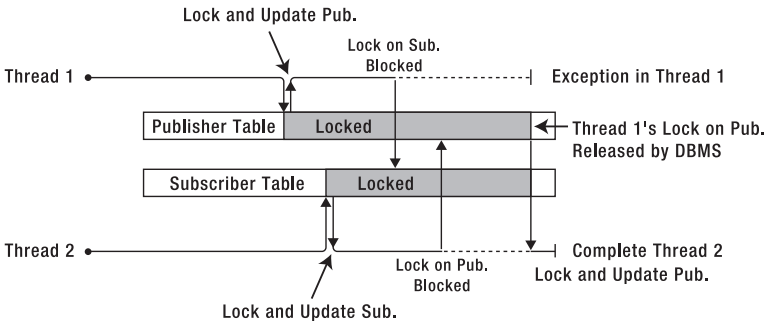

Figure 8-2. *The anatomy of a deadlock*

Looking at the database after completion, you will see that the `test` user has been replaced with either `jeff` or `dave` in both tables (you will never see `dave` from one thread and `jeff` from the other). Though it is not necessary here, because we close the session regardless, in a more extensive application it is important to ensure that the session associated with a deadlock or any other Hibernate exception is closed and never used again because the cache may be left in a corrupted state.

It is worth building and running Listing 8-3 to ensure that you are familiar with the symptoms of a deadlock when they occur.

Listing 8-3. *Code to Generate a Deadlock*

```
package com.hibernatebook.session.deadlock;

import java.sql.Connection;
import java.sql.SQLException;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class GenerateDeadlock {

   private static SessionFactory factory = new Configuration().configure()
         .buildSessionFactory();

   public static void createUser(String username) throws HibernateException {
      Session session = factory.openSession();
      try {
         session.beginTransaction();
```



```
         // Normal usage of the Session here...
         Publisher p = new Publisher(username);
         Subscriber s = new Subscriber(username);
         session.saveOrUpdate(p);
         session.saveOrUpdate(s);

         // Commit the transaction
         session.getTransaction().commit();
      } catch (HibernateException e1) {
         rollback(session);
         throw e1;
      } finally {
         // Close the session
         close(session);
      }
   }

   public static void reset(Session session, int isolation) {
      if (isolation >= 0) {
         try {
            session.connection().setTransactionIsolation(isolation);
         } catch (SQLException e) {
            System.err.println("Could not reset the isolation level: " + e);
         } catch (HibernateException e) {
            System.err.println("Could not reset the isolation level: " + e);
         }
      }
   }

   public static void close(Session session) {
      try {
         session.close();
      } catch (HibernateException e) {
         System.err.println("Could not close the session: " + e);
      }
   }

   public static void rollback(Session session) {
      try {
         Transaction tx = session.getTransaction();
         if (tx.isActive())
            tx.rollback();
      } catch (HibernateException e) {
         System.err.println("Could not rollback the session: " + e);
      }
   }
```

```
public static void main(String[] argv) {

   System.out.println("Creating test user...");
   createUser("test");

   System.out.println("Proceeding to main test...");
   Session s1 = factory.openSession();
   Session s2 = factory.openSession();

   try {
      s1.beginTransaction();
      s2.beginTransaction();

      System.out.println("Update 1");
      Query q1 = s1.createQuery("from Publisher");
      Publisher pub1 = (Publisher) q1.uniqueResult();
      pub1.setUsername("jeff");
      s1.flush();

      System.out.println("Update 2");
      Query q2 = s2.createQuery("from Subscriber");
      Subscriber sub1 = (Subscriber) q2.uniqueResult();
      sub1.setUsername("dave");
      s2.flush();

      System.out.println("Update 3");
      Query q3 = s1.createQuery("from Subscriber");
      Subscriber sub2 = (Subscriber) q3.uniqueResult();
      sub2.setUsername("jeff");
      s1.flush();

      System.out.println("Update 4");
      Query q4 = s2.createQuery("from Publisher");
      Publisher pub2 = (Publisher) q4.uniqueResult();
      pub2.setUsername("dave");
      s2.flush();

      s1.getTransaction().commit();
      s2.getTransaction().commit();

   } catch (RuntimeException e1) {
      e1.printStackTrace();
      // Run the boilerplate to roll back the sessions
      rollback(s1);
      rollback(s2);
      throw e1;
   } finally {
```



```
      // Run the boilerplate to close the sessions
      close(s1);
      close(s2);
   }
  }
}
```

Caching

Accessing a database is an expensive operation, even for a simple query. The request has to be sent (usually over the network) to the server. The database server may have to compile the SQL into a query plan. The query plan has to be run and is limited largely by disk performance. The resulting data has to be shuttled back (again, usually across the network) to the client, and only then can the application program begin to process the results.

Most good databases will cache the results of a query if it is run multiple times, eliminating the disk I/O and query compilation time; but this will be of limited value if there are large numbers of clients making substantially different requests. Even if the cache generally holds the results, the time taken to transmit the information across the network is often the larger part of the delay.

Some applications will be able to take advantage of in-process databases, but this is the exception rather than the rule—and such databases have their own limitations.

The natural and obvious answer is to have a cache at the client end of the database connection. This is not a feature provided or supported by JDBC directly, but Hibernate provides one cache (the first-level, or L1, cache) through which all requests must pass. A second-level cache is optional and configurable.

The L1 cache ensures that within a session, requests for a given object from a database will always return the same object instance, thus preventing data from conflicting and preventing Hibernate from trying to load an object multiple times.

Items in the L1 cache can be individually discarded by invoking the `evict()` method on the session for the object that you wish to discard. To discard all items in the L1 cache, invoke the `clear()` method.

In this way, Hibernate has a major advantage over the traditional JDBC approach: with no additional effort from the developer, a Hibernate application gains the benefits of a client-side database cache.

Figure 8-3 shows the two caches available to the session: the compulsory L1 cache, through which all requests must pass, and the optional level-two (L2) cache. The L1 cache will always be consulted before any attempt is made to locate an object in the L2 cache. You will notice that the L2 cache is external to Hibernate; and although it is accessed via the session in a way that is transparent to Hibernate users, it is a pluggable interface to any one of a variety of caches that are maintained on the same JVM as your Hibernate application or on an external JVM. This allows a cache to be shared between applications on the same machine, or even between multiple applications on multiple machines.

In principle, any third-party cache can be used with Hibernate. An `org.hibernate.cache.CacheProvider` interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation. The cache provider is then specified by giving the implementation class name as the value of the `hibernate.cache.provider_class` property.

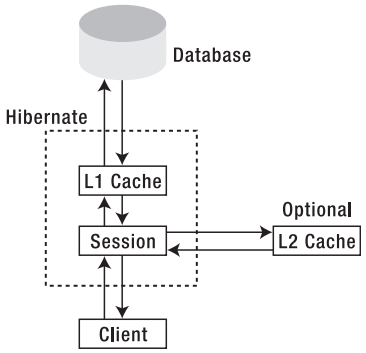

Figure 8-3. *The session's relationship to the caches*

In practice, the four production-ready caches, which are already supported, will be adequate for most users (see Table 8-4).

Table 8-4. *L2 Cache Implementations Supported by Hibernate Out of the Box*

Cache Name	Description
`EHCache`	An in-process cache
`OSCache`	An alternative in-process cache
`SwarmCache`	A multicast distributed cache
`TreeCache`	A multicast distributed transactional cache

The type of access to the L2 cache can be configured on a per-session basis by selecting a `CacheMode` option (see Table 8-5) and applying it with the `setCacheMode()` method.

Table 8-5. `CacheMode` *Options*

Mode	Description
`NORMAL`	Data is read from and written to the cache as necessary.
`GET`	Data is never added to the cache (although cache entries are invalidated when updated by the session).
`PUT`	Data is never read from the cache, but cache entries will be updated as they are read from the database by the session.
`REFRESH`	This is the same as `PUT`, but the `use_minimal_puts` Hibernate configuration option will be ignored if it has been set.
`IGNORE`	Data is never read from or written to the cache (except that cache entries will still be invalidated when they are updated by the session).

The `CacheMode` setting does not affect the way in which the L1 cache is accessed.

The decision to use an L2 cache is not clear-cut. Although it has the potential to greatly reduce access to the database, the benefits depend on the type of cache and the way in which it will be accessed.

A distributed cache will cause additional network traffic. Some types of database access may result in the contents of the cache being flushed before they are used—in which case, it will be adding unnecessary overhead to the transactions.

The L2 cache cannot account for the changes in the underlying data, which are the result of actions by an external program that is not cache-aware. This could potentially lead to problems with stale data, which is not an issue with the L1 cache.

In practice, as with most optimization problems, it is best to carry out performance testing under realistic load conditions. This will let you determine if a cache is necessary and help you select which one will offer the greatest improvement.

Threads

Having considered the caches available to a Hibernate application, you may now be concerned about the risk of a conventional Java deadlock if two threads of execution were to contend for the same object in the Hibernate session cache.

In principle, this is possible—and unlike database deadlocks, Java thread deadlocks do not time out with an error message. Fortunately, there is a very simple solution:

Patient: Doctor, it hurts when I do this.

Doctor: Don't do that then.

Do not share the `Session` object between threads. This will eliminate any risk of deadlocking on objects contained within the session cache.

The easiest way to ensure that you do not use the same `Session` object outside the current thread is to use an instance local to the current method. If you absolutely must maintain an instance for a longer duration, maintain the instance within a `ThreadLocal` object. For most purposes, however, the lightweight nature of the `Session` object makes it practical to construct, use, and destroy an instance, rather than to store a session.

Summary

In this chapter, we have discussed the nature of `Session` objects and how they can be used to obtain and manage transactions. We have looked at the two levels of caching that are available to applications, and how concurrent threads should manage sessions.

In the next chapter, we discuss the various ways in which you can retrieve objects from the database. We also show you how to perform more complicated queries against the database using HQL.